



UNIVERSITATEA DE VEST DIN TIMIȘOARA
FACULTATEA DE MATEMATICĂ ȘI INFORMATICĂ
PROGRAMUL DE STUDII DE LICENȚĂ: Informatică

LUCRARE DE LICENȚĂ

COORDONATOR:
Prof. Dr. Daniela Zaharie

COLABORATOR:
Drd. Alexandru Meca

ABSOLVENT:
Negrea Alexandru Cristian

TIMIȘOARA

2021

UNIVERSITATEA DE VEST DIN TIMIȘOARA
FACULTATEA DE MATEMATICĂ ȘI INFORMATICĂ
PROGRAMUL DE STUDII DE LICENȚĂ: Informatică

Dezvoltarea aplicațiilor WebGIS prin utilizarea bazelor de date cu suport spațial.

COORDONATOR:

Prof. Dr. Daniela Zaharie

COLABORATOR:

Drd. Alexandru Meca

ABSOLVENT:

Negrea Alexandru Cristian

TIMIȘOARA
2021

Abstract

The aim of this bachelor's thesis is to design a Web Feature Service (a REST API) to be used as a tool for spatial data processing, extracting information about given locations, calculate distances between two locations etc. This bachelor's thesis is preponderantly practical because it shows how a software application is made to solve a problem and also how the Smart City idea can be implemented in such a way it can be useful for the client and also for the end-user. Advanced characteristics of programming languages - such as Java, and the popular framework Spring Boot - and the libraries available in Java will be used, as well as programming techniques. The thesis will provide functionality for a web application(for a front-end app) to utilize it's facilities such as : extract data about a given institution type like : medical, public etc, get specific institution by a given name/id, extract locations from given zone with a given radius, the closest location from a given point etc.

Cuprins

Introducere	5
1 Descrierea problematicii	6
1.1 Descrierea problematicii temei alese	6
1.2 Soluții și abordări similare	10
1.3 Publicarea aplicației în cloud AWS	13
2 Arhitectura Aplicației	15
2.1 Structura generală a aplicației	15
2.2 Analiza arhitecturii	17
2.3 Validarea și Tratarea Excepțiilor	25
2.4 Motivare tehnologii alese	28
3 Scenarii de utilizare	30
3.1 Descrierea scenariilor de utilizare	31
3.1.1 User Current Location	33
3.1.2 User Selected Location	33
3.1.3 Get Zone Locations și Get Locations	34
3.1.4 Get Preferred Locations	36
3.1.5 Calculate Distance	37
3.2 Scenarii reale de utilizare	38
4 Concluzii și probleme deschise	40

Introducere

Așa cum se poate observa și în titlul lucrării de licență tema este dezvoltarea aplicațiilor WebGIS prin utilizarea bazelor de date cu suport spațial. Obiectivul principal al lucrării de licență pentru tema specificată mai sus îl reprezintă utilizarea de tehnologii specifice și implementarea unei aplicații software pentru prelucrarea datelor spațiale. Aplicația este un server-based Web Feature Service (WFS) care manipulează informații de tipul geografic cum ar fi : puncte, linii, poligoane etc, aceasta permitând operații de tip query basic CRUD pe date spațiale prin intermediul Web-ului.

Această aplicație, care va fi bazată pe microservicii cu comunicare decuplată între componente(micro serviciile) spațiale prin implementare integrală pe infrastructură cloud AWS, se va ocupa cu returnarea de locații de interes pentru utilizatorii dintr-un oraș anume pe ideea de “SmartCity“ unde aceștia pot primi informații precum: cele mai apropiate instituții medicale, spații publice, rețele de transport etc. De exemplu un utilizator poate să identifice locuri de muncă care sunt cele mai apropiate de locuința sa sau locuințe care sunt foarte aproape de locul de muncă. Aceste locații vor putea fi vizibile prin intermediul API-ului de la “Google“ (Google Maps) pentru a avea o vedere cât mai detaliată și amanunțită a distanțelor și a împrejurimilor.

Capitolul 1

Descrierea problematicii

În acest capitol se va discuta despre descrierea problematicii temei alese, soluții și abordări similare unde se vor exemplifica caracteristici și diferențe cât și similarități dintre aplicațiile deja existente și soluția creată de mine, descrierea instrumentelor pentru rezolvarea problematicii etc.

1.1 Descrierea problematicii temei alese

Așa cum s-a precizat și în partea de introducere a lucrării alese, principalul scop al aplicației îl reprezintă dezvoltarea aplicațiilor WebGIS prin utilizarea bazelor de date cu suport spațial ce utilizează WFS (Web Feature Service).

- **WFS [4]** reprezintă o schimbare a modului în care informațiile geografice sunt create, modificate și schimbate pe internet. În loc să partajeze informații geografice la nivel de fișier folosind protocolul de transfer de fișiere (FTP), de exemplu, WFS oferă acces direct cu informații fine la informații geografice la nivel de caracteristică și proprietate a caracteristicii.

Acesta funcționează prin trimiterea de cereri ce conțin descrieri despre query-ul dorit cât și tipul de transformare de informații ce trebuie efectuate. Request-ul trebuie să fie generat pe partea de client (Web Client) și trimis către aplicația server-based care este până la urmă un Rest API. Aceasta va citii datele și le va executa corespunzător returnând rezultatul dorit.

Specificația WFS necesită ca serverele să accepte un minim de criptare GML, deoarece formatul implicit permite totuși implementatorilor să accepte mai multe formate utile, mai degrabă ca GML, aceste ieșiri vor fi filtrate, paginate și vor avea propriile proprietăți desemnate. Unele dintre aceste encoding-uri este GeoJSON.

GeoJSON [5] este un format de schimb de date geospațiale bazat pe JavaScript Object-Oriented (JSON). Aceasta definește diferite tipuri de obiecte JSON și modul în care sunt combinate pentru a reprezenta datele caracteristicilor geografice, proprietățile și extinderea spațială a acestora. GeoJSON folosește un sistem de referință de coordonate geografice, "World Geodetic System" 1984 și unitățile de grade zecimale.

Formatul se referă la date geografice în sens larg; orice cu calitate care sunt delimitate geografic ar putea fi o caracteristică indiferent dacă este o structură fizică.

Concepțele din GeoJSON nu sunt noi; sunt derivate din preexistente deschise standardele sistemului de informații geografice și au fost simplificate cel mai potrivit pentru dezvoltarea aplicațiilor web folosind JSON. Un exemplu de format GeoJSON ar fi:

```
1  {
2      "type": "Feature",
3      "geometry": {
4          "type": "Point",
5          "coordinates": [125.6, 10.1]
6      },
7      "properties": {
8          "name": "Dinagat Islands"
9      }
10 }
```

Acest standard este unul care respectă și standardul oficial OGC, astfel se poate integra ușor în tipul formatului fără a încălca principiile OGC. Folosind acest standard, aplicația va putea comunica cu multiple dipozitive folosind acest format.

WebGIS este o formă avansată de sistem geospațial de informații (cunoscut ca Sistemul Geografic de Informații) care este disponibil pe platforma web. Acest schimb de informații se realizează între un server GIS (Geospatial Information System) și un client care este reprezentat de un web browser, aplicație mobilă sau aplicație de desktop.

- **GIS:** este un sistem computer-based geografic de informații folosit pentru colectarea, stocarea, manipularea și analizarea datelor spațiale, cum ar fi datele care pot fi referențiate la o locație particulară pe pământ.
- **Spatial data:** cunoscut ca și geospatial data, este o informație despre obiectul fizic care poate fi reprezentat de valori numerice într-un sistem de coordonate ce este reprezentat de: numere ce reprezintă latitudine și longitudinea în format 2D.

Obiectivul principal al acesteia îl reprezintă crearea unei aplicații server based bazate pe microservicii unde prin legarea acesteia de un web client, utilizatorul poate să realizeze multiple operațiuni similare cu aplicația celor de la Google, **Google Maps**. Comunicarea dintre cele două se va realiza prin request-uri HTTP. Prin header folosind “GET”, dar și “POST” pentru trimiterea de obiecte în format JSON.

Aplicația implementată este bazată pe microservicii a căror avantaje și funcționalități vor fi descrise în următoarele capitole. Prin intermediul acesteia, un utilizator prin cadrul unei interfețe web/aplicații de telefon, ar putea obține informații despre o

locație specifică de pe hartă, să caute o anumită locație în funcție de numele acesteia, să calculeze distanța dintre două locații date, să obțină cea mai apropiată locație de acesta în funcție de instituția dorită, să obțină toate locațiile de același tip din acea zonă sau să obțină toate tipurile de locații din raza dată. Aceste funcționalități vor fi mai detaliate în capitolul 3, **Facilități Aplicație**. În esență, această aplicație este creată pentru implementarea ideei de “SmartCity“.

SmartCity: este un framework, predominant alcătuit din Informații și Tehnologii de Comunicare (ICT: Information and Communication Technologies), pentru a dezvolta, implementa și promova practici de dezvoltare durabilă pentru a aborda provocările în creștere ale urbanizării. O mare parte a framework-ului ICT este esențial un network intelligent de obiecte conectate și mașinării (cunoscute ca și orașe digitale) care transmit date folosind tehnologii wireless și cloud.

Aplicațiile IoT bazate pe cloud primesc, analizează și gestionează datele în timp real pentru a ajuta municipalitățile, întreprinderile și cetățenii să ia decizii mai bune care îmbunătățesc calitatea vieții. După mai multe statistică s-au identificat 4 categorii de smart-city care sunt cele mai utilizate, acestea fiind:

- **Essential Services Model:** Orașele din categoria Essential Services Model se caracterizează prin utilizarea rețelelor mobile în programele lor de gestionare a situațiilor de urgență și prin serviciile lor digitale de asistență medicală. Aceste orașe, care ar putea avea deja infrastructuri de comunicații bune, preferă să-și pună banii în câteva programe de oraș intelligent bine alese. Exemplele includ Tokyo și Copenhaga.
- **Smart Transportation Model:** Orașele Smart Transportation Model cuprind cele dens populate și care se confruntă cu probleme de deplasare a mărfurilor și a persoanelor în oraș. Orașele din acest grup subliniază inițiativele de control al congestiunii urbane - prin intermediul transportului public intelligent, partajării mașinilor și / sau autoturismelor - precum și utilizării tehnologiilor informației și comunicațiilor. Singapore și Dubai sunt incluse în această categorie.
- **Broad Spectrum Model:** Orașele care intră sub incidența Broad Spectrum Model subliniază serviciile urbane, cum ar fi gestionarea apei, a apelor uzate și a deșeurilor și căută soluții tehnologice pentru controlul poluării. De asemenea, acestea se caracterizează printr-un nivel ridicat de participare civică. Exemplele includ Barcelona, Vancouver și Beijing.
- **Business Ecosystem Model:** Urmărește să utilizeze potențialul tehnologiilor informației și comunicațiilor pentru a începe activitatea economică. Include orașe care pun accentul pe formarea în competențe digitale ca acoperirea necesară pentru a crea o forță de muncă instruită și care vizează să încurajeze

întreprinderile de înaltă tehnologie. Amsterdam, Edinburgh și Cape Town sunt exemple.

Implicarea WebGIS în crearea ideii de Smart City îl reprezintă un sistem centralizat de informații bazat pe GIS oferă un cadru IT pentru întreținerea și implementarea datelor și aplicațiilor de-a lungul fiecărui aspect al ciclului de viață al dezvoltării orașului.

Exemple de aplicații pentru Smart Cities:

- **Selectia site-ului și achiziționarea terenurilor:** GIS poate combina și integra diferite tipuri de informații pentru a ajuta la luarea unor decizii mai bune și de asemenea oferă, instrumente de vizualizare de înaltă calitate a împrejurimilor care pot imbunătății luarea de decizii. De exemplu: o persoană dorește să achiziționeze un apartament și ar vrea într-o zonă unde este aproape de locul de muncă sau alte puncte de interes. Acesta folosind asemenea aplicații va putea vizualiza în detaliu zone care sunt în interesul acestuia.
- **Planificare, proiectare și vizualizare:** Geodesignul va fi cadrul cheie pentru conceptualizarea și planificarea orașelor inteligente; va ajuta în fiecare etapă, de la conceptualizarea proiectului la analiza amplasamentului, specificațiile de proiectare, participarea și colaborarea părților interesate, crearea proiectelor, simularea și evaluarea.
- **Vânzări și marketing:** Cu GIS, dezvoltatorii orașului pot câștiga potențialele afaceri prin crearea de instrumente informative de vânzări și rapoarte de marketing care evidențiază potențialul economic al unei noi locații sau al dezvoltării viitoare.

De asemenea folosind acest GeoRESTAPI se mai pot folosi: **Open Data** care sunt date ce pot fi folosite în mod liber, refolosite și redistribuite de către oricine și în orice scop, supuse cel mult la necesitatea atribuirii în condiții identice. Astfel se vor putea face previzualizare a datelor spațiale ținând cont de standardele OGC și prin API-ul aplicații se vor pune la dispoziție ca un corp platformă în sprijinul de dezvoltare de aplicații care să consume aceste date.

- **OGC:** [6] este o organizație internațională de standardizare a consensului voluntar, ce a luat naștere în 1994. În OGC, peste 500 de organizații comerciale, guvernamentale, nonprofit și de cercetare din întreaga lume colaborează într-un proces de consens care încurajează dezvoltarea și implementarea standardelor deschise pentru conținutul geospatial și servicii, senzori web și Internetul Obiectelor (Iot - Internet of Things), prelucrarea datelor GIS și partajarea acestora.

Aceasta organizație a standardizat mai multe elemente GIS, unele dintre ele fiind:

- **SRID:** o identificare pentru sistemul de coordonare spațial
- **WMS:** Web Map Service oferă imagini de pe hartă, acesta reprezentând aplicațiile Web ce utilizează server-based applications ce implementează serviciile WebGIS, câteva exemple ar fi: Google Maps, Bing Maps etc.
- **WFS:** Web Feature Service ce sunt folosite pentru recuperarea sau modificarea descrierilor caracteristicilor
- etc

1.2 Soluții și abordări similare

Soluțiile existente pe piată sunt :

- **ArcGIS REST API [23]** care este un REST API care oferă servicii geospațiale, mapare și administrative. Acesta poate fi folosit ca librărie pentru: iOS, Java, .Net etc. Acest REST API este disponibil pentru o varietate de limbaje de programare cum ar fi : C# (împreună cu framework-ul .Net), JavaScript, Python etc. Pe lângă asta, oferă facilități profesionale și optime cum ar fi : “Basemap and data layers”, “Geocoding”, “Data visualization” și aşa mai departe.
- **Bentley Map Enterprise [24]** este un GIS 3D complet dotat cu capacitate puternică de editare a “environment-ului MicroStation”. Este proiectat pentru a răspunde nevoilor unice ale organizațiilor care mapează, planifichează, proiectează, construiește și operează infrastructura lumii. Are capacitatea de a lucra nativ cu toate resursele comune de date spațiale precum PostGIS, Oracle, Microsoft SQL Server Spatial și multe altele. Bentley Map poate adăuga informații semantice către rețea de realitate 3D. Harta Bentley oferă, de asemenea, un SDK complet pentru dezvoltarea personalizată de aplicații “GIS”.
- **GeoMedia [25]** este o platformă de gestionare “GIS” puternică, flexibilă, care va permite să agregați date dintr-o varietate de surse și să le analizați la unison pentru a extrage informații clare, care pot fi acționate. Oferă acces simultan la date geospațiale în aproape orice formă și le afisează într-o singură vizualizare unitară a hărții pentru procesare, analiză, prezentare și partajare eficientă. Funcționalitatea GeoMedia îl face ideal pentru extragerea de informații dintr-o serie de date care se schimbă dinamic pentru a sprijini luarea de decizii în cunoștința de cauză, mai inteligentă.

- **GeoServer** [26] este un server bazat pe Java care permite utilizatorilor să vizualizeze și să editeze date geospațiale. Folosind standardele deschise stabilite de Open Geospatial Consortium (OGC), “GeoServer“ permite o mare flexibilitate în crearea hărților și partajarea datelor. La fel ca și GeoTools este “Use Free“ și “Open Source“.

Câteva caracteristici ale acestor soluții deja existente pe piată sunt următoarele:

- O caracteristică importantă comună ale acestor soluții o reprezintă featurele ce le oferă tuturor clienților lor cum ar fi: fiabilitatea serviciilor oferite, fiind servicii care sunt în industrie de mulți ani oferă un suport și stabilitate foarte bună pentru clienții acestora cât și un număr mare de feature ce le pot utiliza aceștia
- Posibilitatea de a alege dintr-o gamă largă de planuri de subiecție, planul ce conține serviciile și feature-urile care sunt cele mai utilizate și importante pentru dezvoltarea aplicației tale
- Suport expert și customer service profesional care te poate ajuta imediat cu orice problemă ai avea în utilizarea acestor servicii
- Performanțe excepționale din partea serviciilor oferite
- etc

Totuși, cu toate că aceste aplicații sunt similare ca implementarea descrisă în lucrarea aceasta, sunt câteva diferențe care merită să fie explicate și exemplificate pentru a se putea observa o altă abordare în rezolvarea problemei. Câteva dintre aceste diferențe ar fi:

- Implementarea exemplificată folosește librării “open source“ pentru manipularea datelor și returnarea acestora în funcție de request-ul utilizatorului cum ar fi: GeoTools care este o librărie “open source“ ce îți oferă o multitudine de posibilități de configurare și manipulare a datelor, de asemenea aceasta respectă standardele OGC pentru query-ul de date.
- Respectarea standardelor OGC astfel încât aplicația poate fi folosită de mai multe aplicații WMS care să consume endpoint-urile API-ului creat fără a face modificări în ceea ce privește primirea și transmiterea de date
- Utilizarea de bază de date “open source“ PostgreSQL care folosește ca extensie suplimentară PostGIS ce adaugă suportul pentru manipularea de date spațiale și în același timp respectând standardele OGC, astfel aplicația nu este restricționată de aceste standarde pentru manipularea de operații CRUD atât în memorie folosind GeoTools cât și “PostgreSQL“ cu extensia PostGIS

- O altă diferență importantă, prin care aplicația mea se diferențiază de cele create de competitori, o reprezintă faptul că fiind aplicații cu o vechime mai mare folosesc o arhitectură de tipul **“Monolith”**, iar aplicația exemplificată în lucrare va folosi microservicii care vor permite comunicarea cu device-uri multiple prin formatele standard OGC , dar și comunicarea și colectarea de date prin intermediul **“Internet of Things”**.

Aplicația va fi alcătuită din două microservicii care vor avea funcționalități similare dar metode de implementare diferite. Unul din acestea va utiliza baza de date PostgreSQL cu extensia de geodate spațiale PostGIS, iar celălalt microserviciu va utiliza librăria “GeoTools“. Cum aplicația creată are la bază cele două microservicii care folosesc diferite abordări pentru parsarea și manipularea datelor, majoritatea aplicațiilor existente pe piață folosesc arhitecturi de tip **“Monolith”**, care sunt de tip tradițional, unde toate componentele acesteia sunt puse într-un sigur loc ceea ce aduce la o multitudine de dezavantaje cum ar fi:

- **Scalabilitate:** Cu cât aplicația crește, va fi din ce în ce mai greu de menținut, modulele ar putea avea conflicte în împărțirea resurselor, iar la apariția unui bug/malfunction va fi dificil în identificarea acestuia dacă acesta va ajunge să fie enterprise level grade application.
- **Agilitate:** Cum se observă, în zilele noastre, cazurile de utilizare a business-urilor sunt extrem de agile, procesele de business continuă să evolueze. De exemplu, într-o aplicație bancară, continuă să apară noi reglementări. O aplicație de tip **“Monolith”** se poate acomoda la aceste schimbări, dar la costul de reducere al agilității și al menținării deoarece și dacă am schimba o mică parte în aplicație, toată aplicația trebuie să fie verificată ca să nu existe probleme de dependență, trebuie să fie reîmpachetată și asamblată împreună. Așadar se va reduce substanțial producția iar, la un moment dat, procesul de menținere va fi extrem de dificil.
- **Încalcarea principiului SRP:** “Single Responsibility Principle“ care spune că: fiecare modul, clasă sau funcție într-un program ar trebui să aibă responsabilitatea de a se ocupa de o singură parte a aplicației și ar trebui să încapsuleze acea parte de celelalte componente ale aplicației. Încălcând aceasta regulă, implementarea unei noi funcționalități ar fi destul de dificilă deoarece ar putea exista dependențe complexe și cum codul ar fi **“spaghetti code”** ar reduce productivitatea programatorilor enorm de mult.

Mai jos este descrisă structura generală a unei aplicații de acest tip :

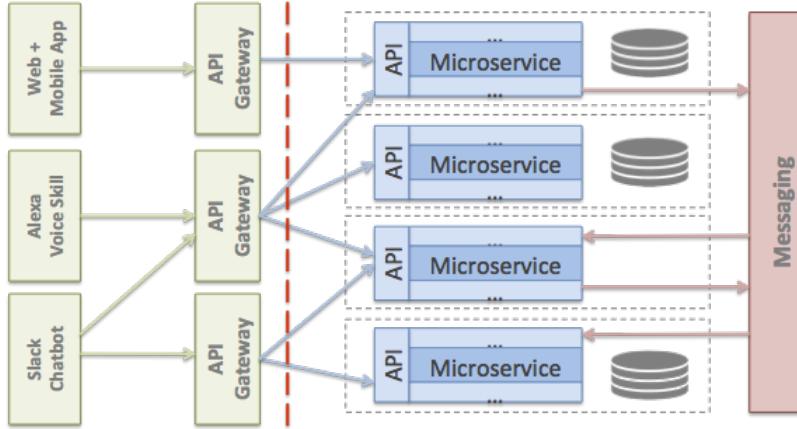


Figura 1.1: Exemplu de arhitectură bazată pe microservicii

1.3 Publicarea aplicației în cloud AWS

Aplicația “sever based“ bazată pe principiul de “WFS“ (Web Feature Service) care este alcătuită din mai multe instanțe de “Spring Boot“ ce alcătuiesc întreaga aplicație se va uploada pe serviciul web oferit de “Amazon“ numit “AWS“ folosind o instanță de “EC2“.

- “Amazon Web Services“ (AWS) este cea mai cuprinzătoare și adoptată pe scară largă platformă cloud din lume, oferind peste 200 de servicii complete de la centrele de date din întreaga lume. Milioane de clienți, inclusiv startup-urile cu cea mai rapidă creștere, cele mai mari companii și agențiile guvernamentale majore folosesc “AWS“ pentru a reduce costurile, pentru a deveni mai agili și pentru a inova mai repede.

Pentru incărcarea atât a aplicației cât și a bazei de date pe platforma “AWS“ a fost nevoie de crearea unui cont pe site-ul celor de la “Amazon“ pentru a utiliza serviciile oferite de aceștia având un “Free Tier“ de utilizare a mai multor servicii de ale lor pe o durată de 12 luni. După crearea acestuia, s-a creat un cloud storage de tipul S3.

- **S3:** [7] “Amazon Simple Storage Service“ (Amazon S3) este un serviciu de stocare a obiectelor care oferă scalabilitate, disponibilitate a datelor, securitate și performanță de vârf în industrie.

“Acet Cloud Storage“ a fost utilizat pentru stocarea JAR-urilor ce reprezintă fiecare instantă “Spring Boot“ necesară pentru utilizarea aplicației. O dată stocate acestea au fost descărcate pe instanța ce va ține aplicația în sine.

Pentru încărcarea și conectarea bazei de date “PostGreSQL“, s-a utilizat “Amazon RDS“ unde s-a creat o instanță specifică bazei de date utilizate după care, folosind credențialele oferite de “Amazon“ s-a făcut o conexiune între baza de date de pe PC cu cea de pe cloud, aceasta fiind accesibilă printr-un “URL“ și “Port Number“.

- **RDS:** [8] “Amazon Relational Database Service“ (Amazon RDS) simplifică configurarea, operarea și scalarea unei baze de date relationale în cloud. Oferă o capacitate scalabilă și rentabilă, automatizând în același timp sarcinile de administrare care necesită mult timp, cum ar fi aprovisionarea hardware, configurarea bazei de date, corecțiile și copiile de rezervă.

Datorită faptului că aplicația “Web Feature Service“ folosește tool-ul “Flyway“ face migrarea bazei de date extrem de ușoară. Acesta când dețează că baza de date la care este concetată aplicația nu coincide cu script-urile “SQL“ care se află în fișierele aplicației, va începe apelarea fiecărui script în parte, astfel reinitializând întreaga baza de date la lansarea aplicației pentru prima dată în noul environment.

În final pentru încărcarea aplicației în sine pe cloud, s-a utilizat serviciul “EC2“ unde s-a creat o instanță de cloud computing cu mai multe resurse (t2.large: 2 nuclee de procesor plus 8 Gib RAM) pentru a putea ține în picioare toate instanțele de “Spring Boot“ într-un mod optim și sigur fără a rămâne fără memorie sau de a întâmpina evenimentul neplăcut numit bottleneck.

- **EC2:** [9] “Amazon Elastic Compute Cloud“ (Amazon EC2) este un serviciu web care oferă o capacitate de calcul sigură și scalabilă în cloud. Este conceput pentru a simplifica calculatorul cloud la scară web pentru dezvoltatori.

S-a încercat utilizarea instanțelor de free tier ale “EC2“ dar, datorită resurselor minimele s-a putut utiliza o singură instanță de “Spring Boot“ pe o instanță de tipul “t2.micro“ (1 nucleu de procesor și 1 Gib RAM). Această abordare nu a funcționat deoarece conexiunea dintre server și microserviciile aplicației era imposibilă, astfel recurgând la varianta descrisă mai sus. Pentru accesarea aplicației se va utiliza un URL de tipul “Public IPv4 DNS“ care va fi utilizat de “Web Map Service“ pentru apelarea endpoint-urilor aplicației cloud.

Capitolul 2

Arhitectura Aplicației

În acest capitol va fi prezentata atât schița arhitecturii cu o ilustrare detaliată unde se vor putea observa elementele aplicației server (back-end) cu, componentele acesteia, descrierea fiecărei componente în parte, modul lor de funcționare, cât și argumentarea folosirii lor.

2.1 Structura generală a aplicației

Aplicația neavând interfață web, arhitectura este orientată server-based ce are ca principale caracteristici două layere de microservicii care au aproximativ aceeași funcționalitate de calcul al datelor spațiale, dar implementarea și modul de funcționare sunt diferite deoarece fiecare dintre acestea cu toate că preiau aceleași tipuri de date care sunt pasate către acestea de o aplicație server modul în care sunt parsate și trimise înapoi către utilizator este diferit.

Aceste date fiind de tip spațial sunt destul de greu de interpretat, iar implementarea ar fi puțin dificilă, de aceea va fi folosită o aplicație de tip server care, în funcție de conținutul datelor, pasează mai departe la microserviciul apelat.

Motivația alegerii arhitecturii cu microservicii multiple, care folosește și servicii REST este data de faptul că este o tehnologie modernă și avansată care oferă multiple implementări și funcționalități. În comparație cu majoritatea soluțiilor existente care folosesc arhitecturi de tip **“Monolith”**, această aplicație folosește tehnologii moderne și de actualitate de microservicii.

- Folosirea de **“Aplicație Server”**, care preia datele de la controlere, le analizează și le transmite mai departe către microserviciul corespunzător care prelucră datele și returnează rezultatul cerut.
- Avantajele microserviciilor par suficient de puternice pentru a convinge unii jucători de mari întreprinderi să adopte metodologia. Comparativ cu structuri de proiectare mai monolitice, microserviciile oferă:
 - **Izolarea erorilor:** aplicațiile mai mari pot rămâne în mare parte neafectate de eșecul unui singur modul.
 - **Ușurința de înțelegere:** cu o simplitate suplimentară, dezvoltatorii pot înțelege mai bine funcționalitatea unui serviciu.

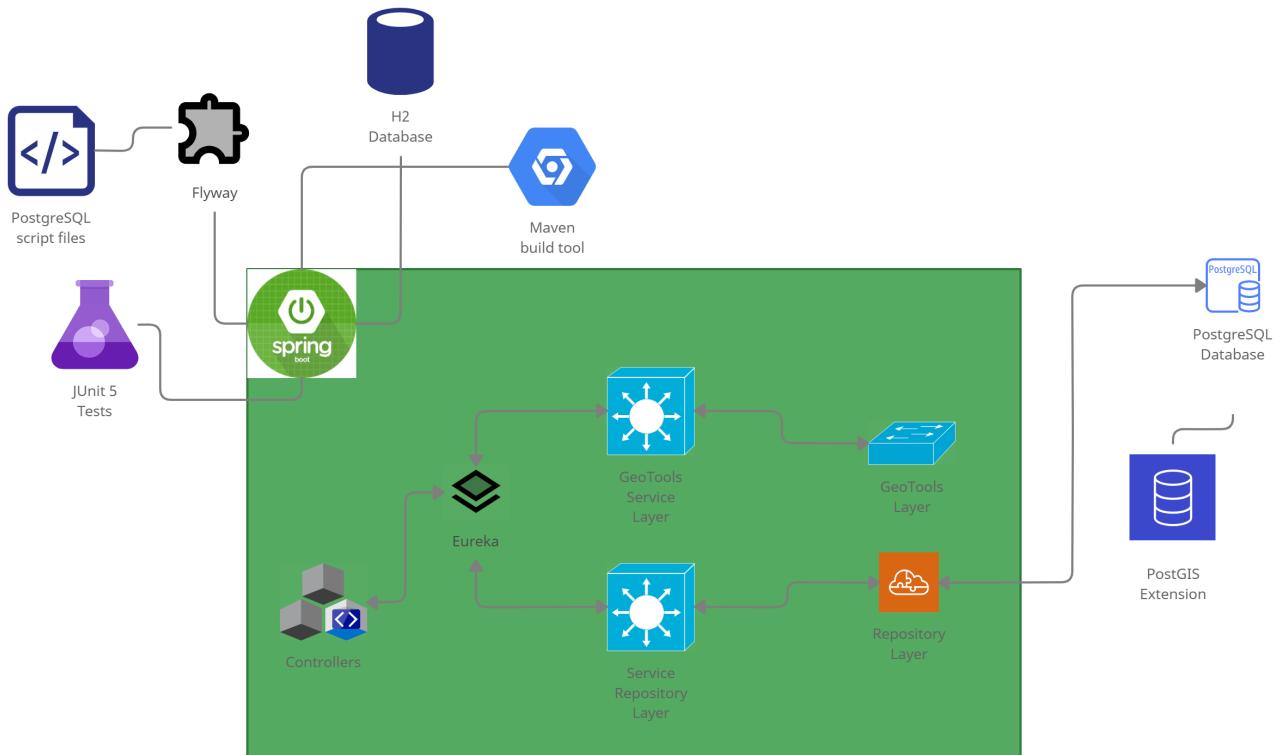


Figura 2.1: Modelul arhitecturii aplicației de tip server ce are la bază microservicii cu multiple layere

- **Implementări mai mici și mai rapide:** baze de cod și scop = implementări mai rapide, care permit, de asemenea, începerea de a explora beneficiile implementării continue.
- **Scalabilitate:** Deoarece serviciile sunt separate, se pot scala mai ușor cele mai necesare servicii la momentele potrivite, spre deosebire de întreaga aplicație. Cand se face corect, acest lucru poate avea un impact asupra economiilor de costuri de performanță.
- Utilizând layere multiple se poate face o distincție clară între activitatea de tip web realizată cel mai bine în controller și logica de afaceri generică care nu este legată de web. Putem testa logica de afaceri legată de servicii separat de logica controllerului. Putem să specificăm comportamentul tranzacției, deci dacă avem apeluri către mai multe obiecte de acces la date, putem specifica că acestea apar în cadrul aceleiași tranzacții.

2.2 Analiza arhitecturii

În această secțiune se va face o analiză mai detaliată a arhitecturilor folosite în aplicație, descrierea motivului folosirii acestora, folosința acestora și modul de operare al aplicației.

Întreaga aplicație va avea ca bază o aplicație server numită “Eureka”[10] care la rândul ei este alcătuită din multiple componente specifice configurării de multiple microservicii ce formează aplicația cloud dorită. Această tehnologie este oferită de **Spring** [11] și este numită **Spring Cloud Netflix** [12].

Componentele vor fi enunțate mai jos explicând modul lor de utilizare, funcționalitatea lor cât și cum sunt conectate unele cu celelalte. Pentru crearea acestei aplicații pe scară largă a fost nevoie de crearea a mai multor instanțe ale framework-ului **Spring Boot** [13] unde fiecare dintre acestea au rolul sau în alcătuirea aplicației în sine. Mai jos vor fi prezentate fiecare dintre aceste module în parte, unde vor fi explicate modul de operare și scopul acestora.

- **WebGISApplicationServer:** Acesta reprezintă modulul principal care ține întreaga aplicație stabilă, monitorizează fiecare integrare de modul în parte, maparea acestora corespunzător și setup-ul server-ului. Configurarea acestui server s-a făcut la nivel de anotări și configurare de proprietăți. Pentru accesarea acestui server unde se pot observa toate instanțele “Spring“ create și un health check al întregii aplicații se poate accesa link-ul : <http://localhost:8661>
- **WebGISApplicationClientRepository:** Modulul acesta reprezintă unul dintre cele două microservicii ale acestei aplicații. Acesta fiind cel care se folosește de baza de date “PostgreSQL“ pentru extragerea datelor și cu extensia acesteia, “PostGIS“, pentru procesarea datelor cu suport spațial. Pentru a putea fi legat de “Aplication Sever“ a fost nevoie de o configurare minuțioasă a acestuia și crearea unei legături stabile cu baza de date. Pe lângă aceste configurații de creare a unei legături cu “Eureka“ a fost implementată cât și :
 - **Flyway:** [14] este un instrument de migrare a bazelor de date open-source. Favorizează puternic simplitatea și convenția față de configurație. Acesta a fost integrat pentru a putea avea un overview asupra scripturilor “.sql“ care se ocupă de crearea de “schemas“, tabelelor necesare, datelor de utilizare cât și a extensilor. Acesta ajută aplicația în cazul unei migrări să creeze tot ce ar avea nevoie ca aplicația să funcționeze optim. În cazul în care ar fi o modificare asupra script-urilor “Flyway“ ar opri aplicația atenționând că a fost făcută o modificare și nu corespunde cu istoricul acestuia.

În continuare acest microserviciu folosește design pattern-ul “Repository Pattern“ care este alcătuit din layere multiple :

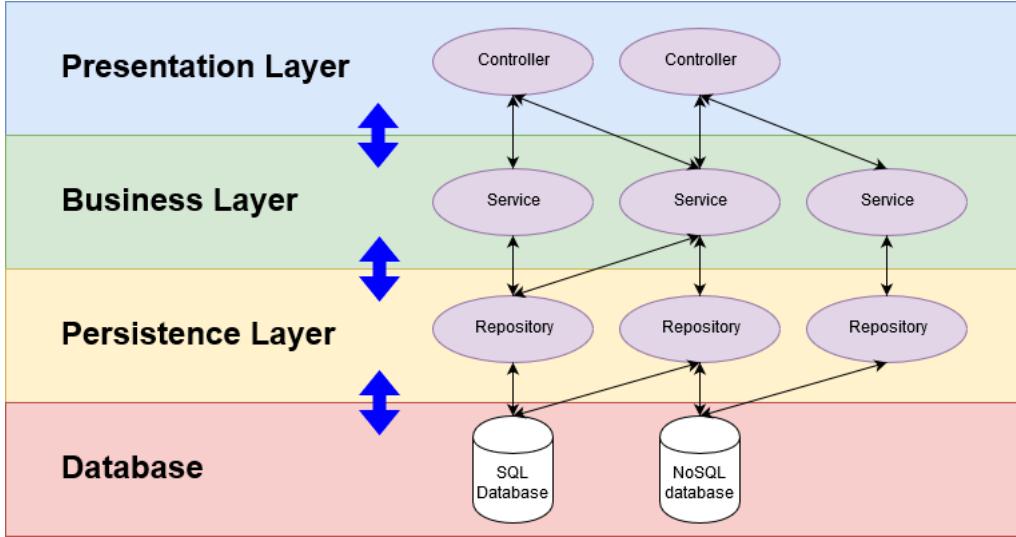


Figura 2.2: Exemplu de implementare a design pattern-ului “Repository Pattern“

- **Service:** Reprezinta layer-ul intermediar dintre “Controller“ si “Repository Layer“ care preia datele de la controller, și de a le da mai departe către repository pentru a putea realiza prelucrarea datelor.
- **Repository:** Layer responsabil cu comunicarea cu baza de date și manipularea datelor primite din partea clientului. Acesta este legat de baza de date “PostgreSQL“ unde trimit diferite **“DDL Statements“** care fac operații cum ar fi : extragerea tuturor locațiilor spitalelor, cea mai scurtă locație dintr-un punct (ex: cea mai scurtă distanță către o instituție medicală), afișarea tuturor locațiilor dintr-o zonă pe o anumită rază. Pe lângă operațiile simple **“CRUD“**, cum baza de date are ca extensie un suport de date spațial **“PostGIS“**, se vor realiza calcule cu aceste date oferind un suport pentru operații cum ar fi: cea mai scurtă distanță către o anumită instituție, distanța dintre două locații etc. Extensia **“PostGIS“** având propriile sale funcții și metode de prelucrare a datelor, oferă sprijin și ajutor în operațiilor cu date de tip spațial. Prin utilizarea anotării **“@Query“** putem folosi asemenea metode. Un exemplu ar fi :

```

1 // TransportInstitutionRepository.java
2 @Query(value = "SELECT buss_stations.id AS
   → id,buss_stations.nume AS nume, "buss_stations.code AS
   → code" + ", buss_stations.latitude AS latitude,
   → buss_stations.longitude AS longitude" + " FROM
   → public.buss_stations buss_stations",nativeQuery = true)
3 List<TransportInstitution> getBusStationLocations();

```

– **Controller:** Reprezintă **Presentation Layer** care se ocupă cu preluarea request-urilor venite din partea de front-web (aplicație web) și transmiterea mai departe la următoarele layere care se vor ocupa cu prelucrarea datelor primite. Acesta este alcătuit din mai multe clase care fiecare reprezintă o funcționalitate diferită pentru aplicația în sine :

- * **TransportLocationController**
- * **MedicalLocationController**
- * **PublicLocationController**
- * **UserController**

La nivelul acestui layer se realizează și validările obiectelor primite în format JSON și parsate la POJO-urile aplicației. În fiecare POJO există validări la nivelul fiecărui field care verifică corectitudinea informațiilor iar, în cazul în care o condiție nu este îndeplinită, acest microserviciu să arunce o eroare înapoi către client. Aceste validări se realizează cu librăria (proiect separat) numit **Validation** [15] și se folosesc utilizând anotare “@Valid“ :

```

1 public List<Institution> getAllLocationsFromZone(@Valid
   → @RequestBody Point point){
2     //implementation
3 }

```

Cum s-a putut observă aplicația folosește **Spring Data JPA** [16] care are la bază “CrudRepository“ ce folosește operații **CRUD**, astfel permitând utilizarea de “Named Queries“ pentru a trimite comenzi către baza de date. Această anotare are la bază metodele “Java“, aşadar se poate lega direct de **Spring Data JPA**.

Pentru aceste “Queries“ s-au creat și utilizat entități pentru preluarea datelor din baza de date. Aceste entități sunt simple “POJOS’s“ care sunt anotate cu “@Entity“ pentru a putea lucra cu proiectul separat : **Spring Data JPA**. Mai trebuie precizat faptul că a mai fost folosită și librăria **Lombok** [17] pentru scrierea a cât mai puțin cod. Aceasta se utilizează folosind anotări precum “@Data“

care creează pentru clasa respectivă, “Getters, Setters etc“, cât și alte anotări care ajută developer-ul să fie cât mai productiv și eficient.

De asemenea mai trebuie menționat faptul că microserviciul este documentat folosind Open API-ul **Swagger** [18]. Folosind anotări corespunzătoare s-au putut documenta fiecare endpoint al aplicației cât și POJO-urile utilizate în implementare. **Swagger** fiind un Open API puternic acesta permite prin intermediul url-ului `http://localhost:8183/swagger-ui.html` vizualizarea tuturor endpoint-urilor cât și testarea acestora. Aceasta se accesează folosind server port-ul microserviciul, astfel putem să deosebim “ClientRepository“ de “ClientGeoTools“.

- **WebGISApplicationClientGeoTools:** Modulul acesta reprezintă celălalt microserviciu al aplicației. Acesta nu se folosește de baza de date “PostgreSQL“ pentru procesarea de date ci utilizează librăria “Open Source“ **GeoTools** care folosește implementări Java pentru prelucrarea și procesarea datelor.
 - **GeoTools** [19] este o bibliotecă de cod Java open source (LGPL) care oferă metode conform standardelor pentru manipularea datelor geospațiale, de exemplu pentru implementarea sistemelor de informații geografice (GIS). Biblioteca GeoTools implementează specificațiile Open Geospatial Consortium (OGC) pe măsură ce sunt dezvoltate. Această librărie este de asemenea “Open Source“ oferind codul sursă către un public “GIT Repository“ și de asemenea este “Open Development“ oferind sprijin companiilor pentru oferirea de release-uri specifice și stabile.

Ca și celălalt microserviciu acesta se leagă de “Application Server“, primul modul prezentat. Configurarea este aceeași, doar ca acesta trebuie să aibă un alt nume al aplicației cât și un alt server port pentru a nu se suprapune cu celălalt microserviciu pentru ca **Eureka** să știe să le diferențieze și să poată să facă legăturile necesare fără probleme. Acest microserviciu implementează doar acele endpoint-uri care folosesc extensia **PostGIS** folosind **GeoTools**, la fel ca și modulul anterior, se folosește “Repository Pattern“, având :

- **Presentation Layer (Controller):** Care se ocupă cu preluarea request-urilor venite din partea de front-web (aplicație web) și transmiterea mai departe la următoarele layere care se ocupă cu prelucrarea datelor primite. Acesta este alcătuit dintr-o singură clasă care se ocupă de acele endpoint-uri ce pot utiliza librăria externă. Aceste endpoint-uri primind datele identice ca la celălalt microserviciu și rezultatele identice în proporție de 90%.

- **Business Layer (Service):** Reprezintă layer-ul intermediar dintre “Controller“ și “Repository Layer“ care preia datele de la controller, și de a le da mai departe către repository pentru a putea realiza prelucrarea datelor.
- **Persistence Layer (Repository):** Este layer-ul principal care se ocupă cu partea de prelucrare și parsare a datelor primite de client. Acesta utilizează librăria “GeoTools“ unde calculează distanța dintre două puncte date, preia locația unei anumite instituții (spital, farmacie) cea mai apropiată de un anumit punct dat cum ar fi locația user-ului, toate locațiile dintr-un anumit punct dat, dar se poate returna și un anumit tip de locații cum ar fi : toate scolile, spitalele etc.

Și acest modul se folosește de simple “POJOS” care sunt anotate cu “@Entity“ pentru a putea lucra cu proiectul separat : “Spring Data JPA“ și sunt utilizate și anotări din proiectul “Lombok“ pentru productivitate și eficiență.

De asemenea și acest microserviciu este documentat cu Open API-ul **Swagger** folosind aceleași anotări atât pentru controller pentru a documenta endpoint-urile cât și “POJO-urile“ folosite în acesta. Pentru a accesa această documentație trebuie accesat url-ul <http://localhost:8184/swagger-ui.html>.

Pentru a diferenția microserviciile se vor utiliza porturi diferite.

- **WebGISApplicationFeignClient:** Aceasta este unul dintre cele două module de **Feign Client** care se ocupă de îmbinarea clientului de servicii web cu modulul de microserviciu numit **WebGISApplicationClientRepository**, astfel putem să legăm microserviciul respectiv prin modulul de **Feign**, iar un alt modul **ZuulProxy** (despre care se va vorbi la urmă) care va face legăturile dintre microservicii și feign client-urile lor respective. Acestea nu fac altceva decât să paseze mai departe datele către microserviciul corespunzător. Configurarea unui asemenea client este destul de simplă, tot ce este nevoie este să oferim un nume distinct cât și un port pentru a putea fi înregistrat de “Eureka“. Un **Feign Client** este alcătuit dintr-un controller, serviciu și “POJO-urile“ ce le utilizează.
 - **ClientResourceController:** Clasa aceasta se comportă ca un “RestController“ obișnuit, dar ca resurse folosește o instanță a unei interfețe service numită **ClientServiceFeign** care are ca anotare “`@FeignClient(name = "web-gis-client-repository")`“, unde “name“ reprezintă numele microserviciului pe care îl folosește ca resursă.

Această configurare de “Resource Controller“ arată așa :

```
1  @Autowired
2  public ClientResourceController(ClientServiceFeign
3      ↪ clientServiceFeign){
4      this.clientServiceFeign = clientServiceFeign;
5      ↪ //resursa de feign client
6
7  }
8
9  @ResponseBody
10 @GetMapping("/public/locations/get") // endpoint-ul complet
11 List<PublicInstitution> getAllPublicLocations(){
12     return clientServiceFeign.getAllPublicLocations(); // 
13     ↪ apelarea resursei
14 }
```

```

11
12 //code continues here

```

- **ClientServiceFeign:** Reprezintă interfața ce este îmbinată cu microserviciul “WebGISApplicationClientRepository“ și prin intermediul proxy-ului se transmit datele către acest modul.

Iar acea configurare de “Feign Client“ arată aşa :

```

1 @FeignClient(name = "web-gis-client-repository") // legatura
   ↵ cu microserviciul
2 public interface ClientServiceFeign {
3
4 @ResponseBody
5 @GetMapping("/public/locations/get")
6 List<PublicInstitution> getAllPublicLocations();
7
8 //code continues here

```

- **WebGISApplicationFeignClientGeoTools:** Este ultimul din cele două module de **Feign Client** care se ocupă de îmbinarea clientului de servicii web cu modulul de microserviciu numit **WebGISApplicationClientGeoTools**, astfel putem să legăm microserviciul respectiv prin modulul de **Feign**, iar un alt modul **ZuulProxy** care va face legăturile dintre microservicii și feign client-urile lor respective. Ca și modulul “Feign Client“ anterior, are aceleași implementări și configurații.
 - **ClientGeoToolsResourceController:** Clasa aceasta se comportă ca un “RestController“ obișnuit, dar ca resurse folosește o instanță a unei interfețe serviciu numită **ClientServiceFeign** care are ca anotare “`@FeignClient(name = "web - gis - client - geo - tools")`“, unde “name“ reprezintă numele microserviciului pe care îl folosește ca resursă.
 - **ClientGeoToolsServiceFeign:** Reprezintă interfața ce este îmbinată cu microserviciul “WebGISApplicationClientGeoTools“ și prin intermediul proxy-ului se transmit datele către acest modul.
- **WebGISApplicationZuulProxy:** Acesta reprezintă un modul (o instanță de “Spring Boot“) care are rolul de a face legăturile dintre microserviciile enunțate și detaliate mai sus și feign client’s care ajută la scrierea de servicii web într-un mod mai elaborat și mai ușor. Configurarea este realizată la nivel de proprietăți și ar arăta în felul următor: fiecare feign client reprezintă un resource service pentru microserviciul lui respectiv care este un client service. Accesarea acestor

microservicii se va realiza pe același port, dar pentru a deosebi ce modul folosim, endpoint-ul de accesare este diferit pentru fiecare : /api/* este pentru “web-gis-client-repository“ iar, /geo-tools/** pentru “web-gis-client-geo-tools“. Această configurare arată în felul următor :

```
1  zuul:
2      host:
3          connect-timeout-millis: 5000000
4          socket-timeout-millis: 5000000
5          ignoredServices: '*'
6      routes:
7          geo-tools-resource-service:
8              path: /api/geo-tools/**
9              serviceId: web-gis-application-feign-client-geo-tools
10             stripPrefix: true
11         resource-service:
12             path: /api/**
13             serviceId: web-gis-application-feign-client
14             stripPrefix: true
15         user-service:
16             path: /repository/**
17             serviceId: web-gis-client-repository
18             stripPrefix: true
19         geo-tools-user-service:
20             path: /geo-tools/**
21             serviceId: web-gis-client-geo-tools
22             stripPrefix: true
```

Datele primite vor fi în format “**JSON**“(JavaScript Object Notation) prin intermediul protocolelor “**HTTP**“. Cele mai întâlnite protocole “**HTTP**“ sunt următoarele:

- **GET**: folosit pentru extragerea de informații din baza de date.
- **POST**: utilizat pentru salvarea datelor în cadrul bazei de date.
- **PUT**: folosit pentru actualizarea datelor existente.
- **DELETE**: folosit pentru ștergerea datelor.

2.3 Validarea și Tratarea Exceptiilor

În această secțiune se va discuta atât despre validarea datelor provenite din partea de front-end (partea web a aplicației), acestea venind sub format JSON, cât și despre tratarea exceptiilor care sunt de formă: locația după nume/id nu este gasită, obiectul JSON are parametrii lipsă sau incorecti etc.

- **Validări:** acestea au un rol foarte important în asigurarea stabilității aplicației server cât și de corectitudine a datelor provenite de la client. De pe partea clientului există posibilitatea să se trimită un bad HTTP request și datele din baza de date să fie avariate, în acest sens s-au luat multiple precauții pentru a asigura validitatea datelor. Aceste validări, provenite din pachetul extern **Hibernate [20]** se folosesc de numeroase anotări care puse deasupra field-urilor unui POJO Java, în cazul nerespectării acelei restricții acesta va arunca o excepție corespunzătoare. Mai jos este un exemplu de acest tip de validare:

```
1 @NotNull(message = "The string name may not be null")
2 @NotEmpty(message = "The string name may not be empty")
3 @NotBlank(message = "The string name may not be blank")
4 @Pattern(regexp="^A-Za-z*$",message = "Invalid Input")
5 private String name;
```

- **@NotNull:** se asigură că field-ul nu va avea o valoare null, deoarece în acest caz s-ar putea returna valori corupte.
- **@NotEmpty:** acesta se asigură ca field-ul nu va fi gol, adică lungimea acestuia să fie mai mare ca zero.
- **@NotBlank:** se asigură ca field-ul să nu fie doar spații goale sau caractere nevizibile și să se trimită un query corrupt către baza de date. Cu toate că tabelele din baza de date nu ar fi afectate și doar s-ar fi returnat empty list sau null, am asigurat să nu se execute algoritmi de parsare a datelor fără sens.
- **@Pattern:** această anotare se asigură că field-ul în funcție de un regex pattern să indeplinească această condiție. În cazul de mai sus textul să contină doar litere.

Pentru toate aceste anotări, în cazul în care condiția nu este indeplinită se va afișa mesajul custom dat mai sus pentru fiecare anotare în parte. Totuși pentru a face ca această validare să funcționeze este nevoie de încă o anotare specială care verifică datele în formatul JSON imediat de la primirea prin endpoint-ul respectiv:

```

1 public List<InstitutionDTO> getAllLocationsFromZone(@Valid
2     ↳ @RequestBody Point point){
3         return userService.getAllLocationsFromZone(point);
4     }

```

- **@Valid:** această anotare activează verificarea datelor prin pachetul extern “Hibernate”, folosind acele anotări exemplificate și explicate mai sus. În cazul unei verificări eşuate o excepție va fi aruncată.

- **Tratarea Excepțiilor:** O excepție este un eveniment care are loc în timpul executării unui program, care perturbă fluxul normal al instrucțiunilor programului. Acestea se manifestă prin aruncarea unui obiect de acest tip numit : **Exception Object** către “runtime system“. Prin tratarea și manipularea excepțiilor putem să asigurăm flow-ul fluid și neîntrerupt al aplicației noastre. Validarea și tratarea excepțiilor sunt combinate pentru a putea crea o strânsă legătura între acestea și tratarea celor mai frecvente și dăunătoare excepții. Mai jos vor fi prezentate excepțiile care se pot realiza în interiorul aplicației.

- **Status: 400 Bad Request:** acesta manifestându-se datorită unui obiect JSON trimis greșit sau incomplet afișând mesajul:

```

1 {
2     "status": "BAD_REQUEST",
3     "message": "Operation not permitted.",
4     "errors": [
5         "Operation not permitted. because something inside the
6             ↳ object or the object was : null"
7     ]

```

- **Status: 404 Not Found:** acest status este declanșat dacă pentru găsirea unei locații după id sau după nume nu a fost găsită se va trimite mesajul de mai jos

```

1 {
2     "status": "NOT_FOUND",
3     "message": "Object not found",
4     "errors": [
5         "Object not found because something inside the object
6             ↳ or the object was : null or doesn't exist."
7     ]

```

- **Status: 406 Not Acceptable:** acest status este declansat dacă tipul de instituție trimis nu este găsit/valabil

```

1  {
2      "status": "BAD_REQUEST",
3      "message": "Operation not permitted.",
4      "errors": [
5          "Operation not permitted. because something inside the
6              → object or the object was : null"
7      ]
8  }

```

Cu toate acestea, validarea și tratarea excepțiilor ce pot veni printr-un endpoint sunt tratate prin crearea de excepții custom și crearea unei clase cu o anotare specifică acestui tip de situație. Pentru afișarea mesajelor de excepții se folosește o clasă custom care preia doar informațiile necesare. Aceste clase custom sunt de forma:

```

1 public class ConstraintViolationExceptionCustom extends
2     → RuntimeException {
3     public ConstraintViolationExceptionCustom(){
4         super("Bad Object Request.");
5     }

```

Iar clasa principală de prinderea și afișarea mesajelor de eroare este de forma următoare:

```

1 @RestControllerAdvice
2 public class GlobalExceptionHandler {
3
4     @ResponseStatus(value = HttpStatus.NOT_FOUND, code =
5         → HttpStatus.NOT_FOUND)
6     @ExceptionHandler(NotFoundException.class)
7     public ResponseEntity<Object> notFoundException(RuntimeException
8         → exception) {
9
10        String error = exception.getMessage() + " because something
11            → inside the object or the object was : " +
12            → exception.getCause() + "" +
13                " or doesn't exist.";
14
15        APIError apiError =

```

```

12         new APIError(HttpStatus.NOT_FOUND,
13             ↵ exception.getLocalizedMessage(), error);
14     return new ResponseEntity<>(apiError, new HttpHeaders(),
15             ↵ apiError.getStatus());
16 }
17 ...

```

Anotarea “@RestControllerAdvice“ se pune deasupra clasei pentru a fi înșințătată în contextul “Spring“ ca și clasa de tratare a excepțiilor pentru endpoint-uri iar, “@ExceptionHandler(NotFoundException.class)“ este folosit pentru a specifica clasei ce tip de eroare așteptăm (în interiorul parantezelor se va specifica tipul de excepție la care ne așteptăm. De reținut este faptul că, pentru fiecare microserviciu și feign client-ul respectiv al acestuia are o asemenea configurație de tratare a excepțiilor.

2.4 Motivare tehnologii alese

În această secțiune se va vorbi în special de motivația alegerii tehnologiilor alese, cât și avantajele acestora. Aplicația propriu-zisă este dezvoltată cu ajutorul unui framework “Spring Boot“, limbajul de programare folosit la baza acestuia este Java. “Spring Boot“ este un micro-cadru open-source întreținut de o companie numită “Pivotal“. Oferă dezvoltatorilor Java o platformă pentru a începe cu o aplicație de “Spring“, configurabilă automat, de producție. Cu acesta, dezvoltatorii pot începe rapid fără a pierde timp la pregătirea și configurarea aplicației lor de Spring. Unele dintre avantajele acestui framework fiind următoarele:

- **Flexibilitate:** Spring suportă atât configurare legacy pe bază de XML cât și “Java-based annotations“ care este de noutate și te ajută pentru configurarea de “Spring Beans“. Aceasta putând să fie rulat cu orice tip de configurare permite crearea de aplicații enterprise foarte ușor.
- **Standalone:** Framework-ul Spring vine la pachet cu un lightweight container. Acesta poate fi activat fără să se folosească un web server sau “application server“. Cu toate acestea aplicația creată folosește o versiune lite de web server “Apache Tomcat“ care este foarte rapid și stabil.
- **Cadrul web bine conceput:** Spring are un framework MVC foarte bine construit care aduce o varietate de alternative la un web framework legacy și de asemenea permite integrarea foarte ușoară a altor pachete pentru extinderea și îmbunătățirea aplicației server web.

În concluzie, Spring este un framework care a ajuns la maturitate, este întreținut și actualizat cu grijă. Este foarte rapid în ceea ce privește atât performanța acestuia cât și viteza de producție a unei aplicații server web.

Capitolul 3

Scenarii de utilizare

Pentru prezentarea facilităților aplicației, am creat o diagramă “**Use Case**“ (diagrama a cazurilor de utilizare) care prezintă acțiunile principale ale aplicației.

Cum lucrarea de licență este axată mai mult pe partea de server-based, diagrama prezentată este construită pentru a ilustra cazurile de utilizare din perspectiva server-ului.

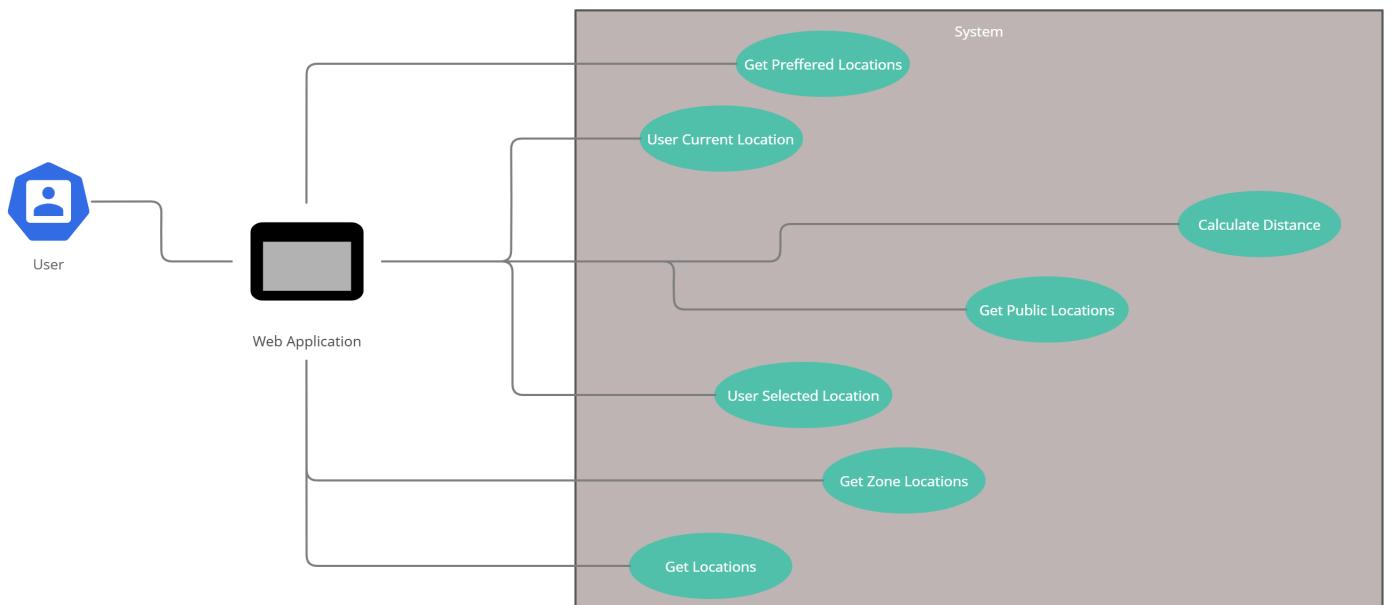


Figura 3.1: Diagrama **Use Case** unde sunt exemplificate câteva din cazurile de utilizare ale aplicației

Mai jos vor fi explicate operațiile efectuate de cazurile de utilizare prezentate mai sus:

- **User Current Location:** Acest use case va returna locația curentă a utilizatorului.
- **User Selected Location:** Aceasta va returna o locație dată de utilizator.
- **Get Zone Locations:** Use case-ul acesta este puțin mai complex deoarece va returna pe baza unei locații date, toate zonele de interes dintr-o anumită zonă.
- **Get Locations:** Acest use case va returna locații în funcție de preferințele clientului pe o anumită rază.

- **Get Preferred Locations:** Va oferi utilizatorului locații în funcție de o anumită categorie arătând drumul cel mai scurt dintr-o anumită zonă selectată de acesta.
- **Calculate Distance:** Așa cum sugerează și numele, în funcție de două locații date de utilizator, aplicația va returna distanța dintre acestea cât și alte date utile pentru acesta.

Pentru crearea unui scenariu de utilizare s-a creat un Web Map Service (WMS) al cărui scop este să exemplifice un mod de utilizare a Web Feature Service-ului (WFS), aplicația prezentată și descrisă în capituloanele de mai sus. Aplicația de WFS a fost creată și realizată utilizând framework-ul popular **Angular** [21].

- **Angular:** este un cadru de proiectare al aplicației și o platformă de dezvoltare pentru a crea aplicații eficiente și sofisticate de o singură pagină.

3.1 Descrierea scenariilor de utilizare

Harta folosită în această aplicație este furnizată de OSM ce folosesc vectori punând pe mapă locațiile dorite, prin acest serviciu se fac call-uri de tipul HTTP GET și primesc tile-uri astfel consumând acest serviciu de la **OpenStreetMap** [22]. Un tile reprezintă grafică bitmap pătrată afișată într-un aranjament grilă pentru a afișa o hartă.

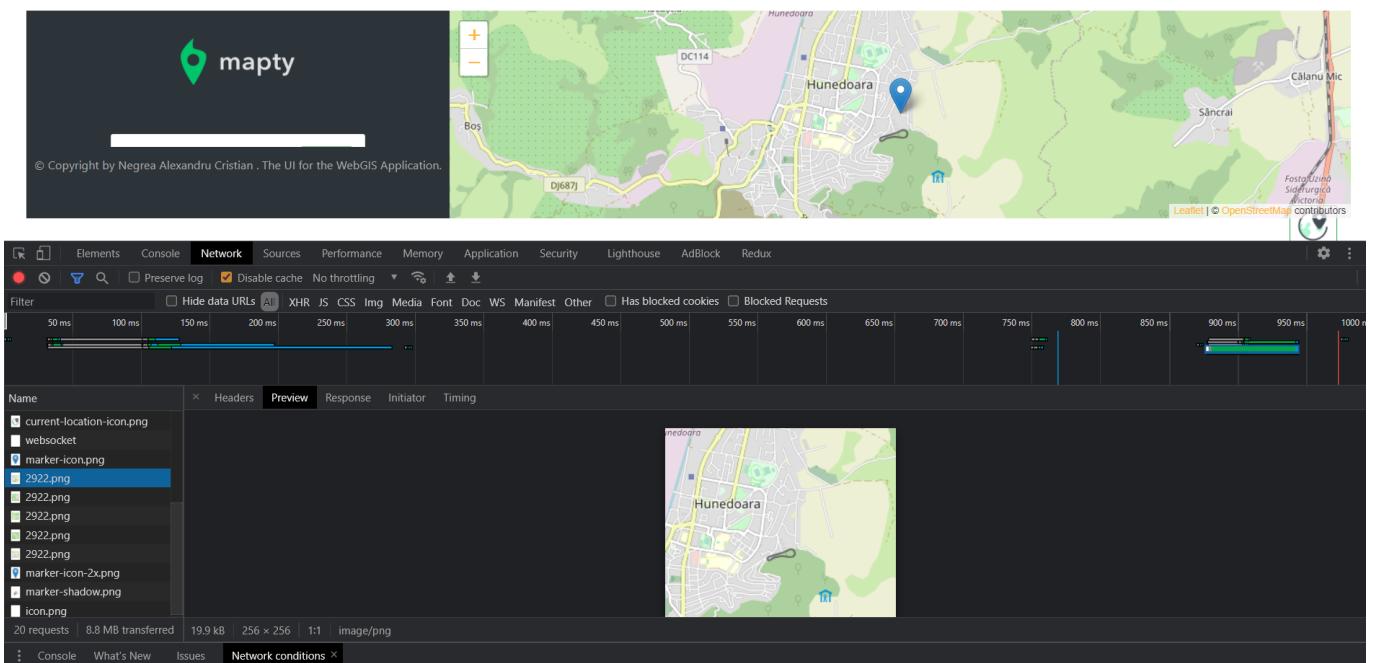


Figura 3.2: Această imagine arată o exemplificare cum se face un request de tipul GET care returnează un vector de tile map-uri.

Un tile dintr-o mapă are deobicei dimensiunea de 256×256 pixeli, dar nu au întotdeauna această dimensiune; de exemplu, pot exista imagini de 64×64 pixeli, cu toate acestea, imaginile de 256×256 pixeli sunt un standard, iar 512×512 pixeli pare a fi dimensiunea normală pentru tile-urile de înaltă rezoluție.

În tabloul de network se poate observa în partea din stânga jos mai multe request-uri de tip GET care pentru fiecare afișare a mapei, apelează acel endpoint care returnează o listă de tile-uri ce conțin câte o bucătă din acea poziție a mapei care se afisează pe ecran. Toată mapa este o imagine de tipul JPG, dar pentru fiecare afișare a unei părți a acesteia se va returna un vector de tile-uri ce reprezintă o bucătă a acelei mape.

Aplicația demonstrativă ce exemplifică o utilizare a serviciului WFS creat are o interfață user-friendly care este usor de utilizat.

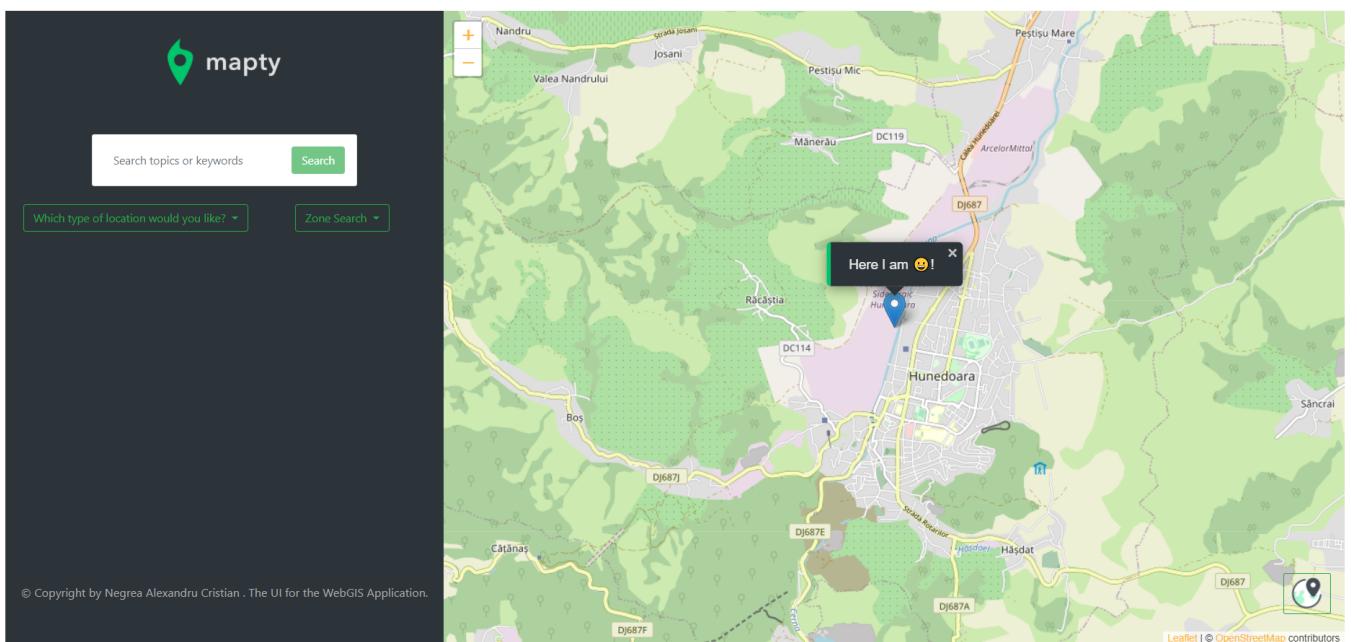


Figura 3.3: Interfața WMS propriu-zisă.

După cum se poate observa și din imaginea de mai sus aplicația are câteva funcționalități ce sunt utile utilizatorului și ce exemplifică use case-urile enumerate la începutul capitolului :

3.1.1 User Current Location

- **User Current Location:** este un use case ce este utilizat pentru afișarea locației curente a utilizatorului.

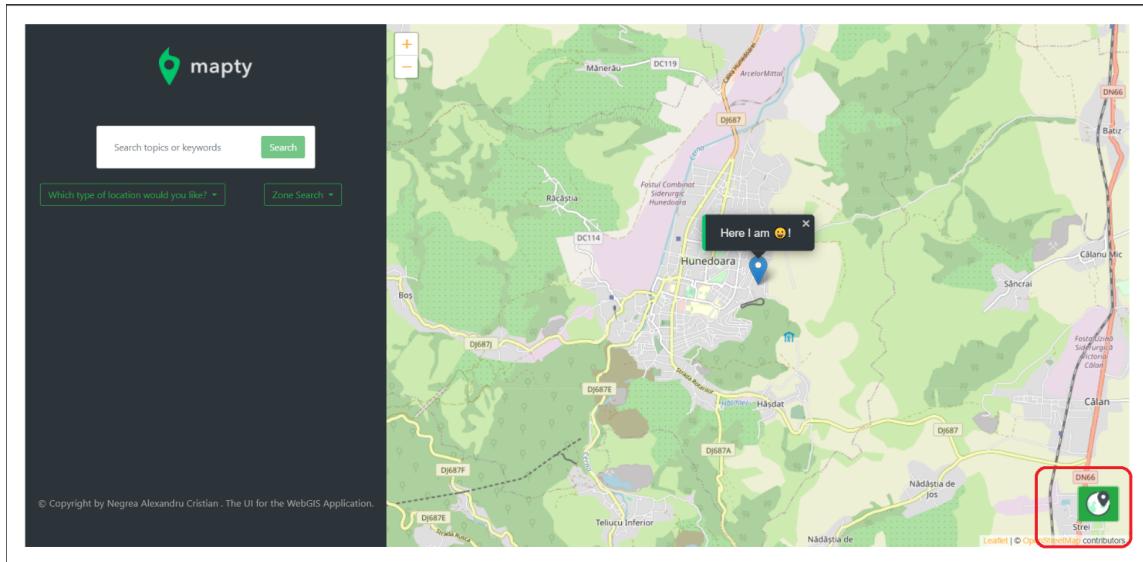


Figura 3.4: Locația curentă.

În chenarul roșu din dreapta jos se află butonul care o dată apăsat va pune un marker pe mapă arătând locația utilizatorului în acel moment cu un mesaj ”Here I am :)!”. Acest use case se realizează prin intermediul interfeței web folosind Web API luând locația curentă a utilizatorului :

```
1      if (navigator.geolocation) {  
2          navigator.geolocation.getCurrentPosition(  
3              this.setGeoLocation.bind(this),  
4              this.handleError.bind(this)  
5          );  
6      }  
7  }
```

3.1.2 User Selected Location

- **User Selected Location:** îi permite utilizatorului să caute după o locație specifică după numele acesteia.

În cele 2 chenare roșii se află următoarele: cel din stânga sus reprezintă textbox-ul unde se poate introduce numele locației dorite. O dată introdusă va apărea un marker ce ilustrază locația căutată ca în chenarul roșu din dreapta. Această acțiune se realizează prin apelarea unui endpoint de pe back-end (aplicația WFS) ce returnează locația după numele dat, în caz că user-ul introduce un nume care

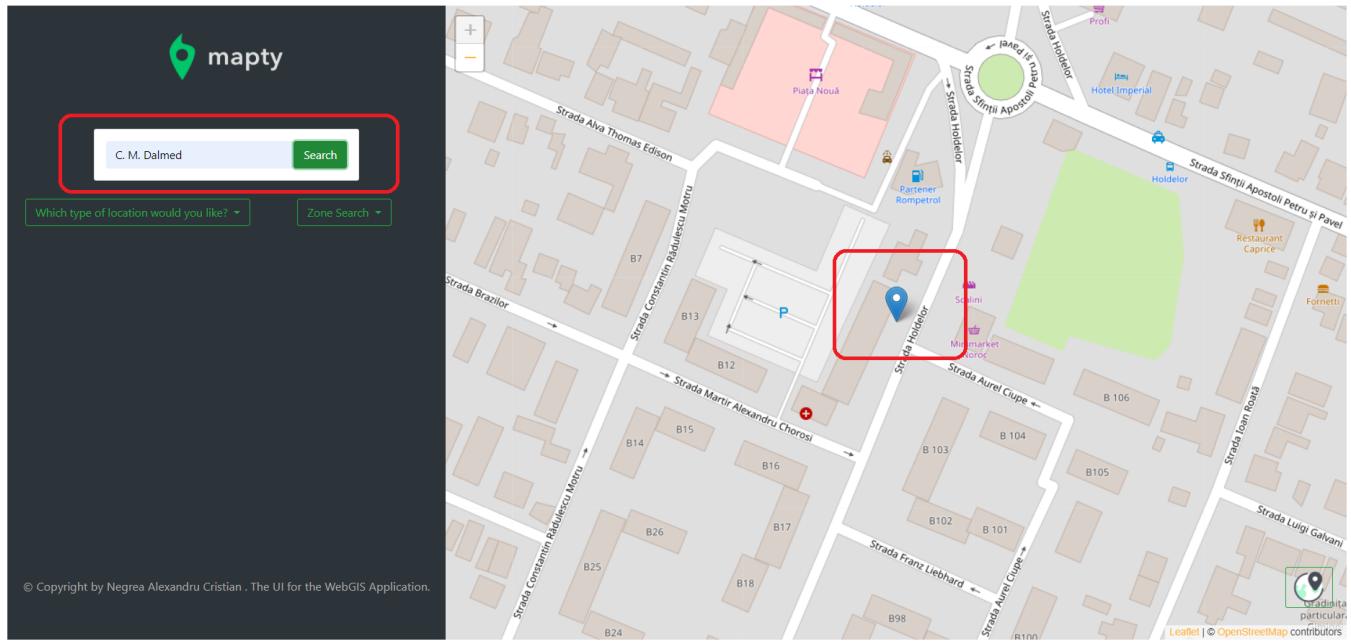


Figura 3.5: Căutarea locației după nume.

nu e 100% corect atunci se va face o căutare parțială bazată pe acel nume. O dată venită locația pe front-end folosind NgRx această locație se va afișa pe mapă indiferent câte markere ar mai exista pe hartă la acel moment. În cazul în care locația căutată de user nu există și acel “partial search” eșuează în a găsi un match pentru text-ul introdus de utilizator atunci se va afișa un alert ce îi va spune utilizatorului că locația introdusă de acesta nu există.

3.1.3 Get Zone Locations și Get Locations

- **Get Zone Locations și Get Locations:** vor returna locațiile din zona din care este user-ul în acel moment (locația sa curentă), sau folosind celălalt microserviciu al aplicației se pot returna locațiile dintr-o zonă selectată de user, fie introducând coordonatele manual.

În cele 2 cadrane roșii se pot observa următoarele: în partea stângă avem meniul aplicației care la apăsarea butonului 'Zone Search' se va deschide un drop-down de unde utilizatorul va putea să aleagă ce tip de 'Zone Search' dorește acesta: 'Zone Search' este și acela utilizat în imagine și returnează toate locațiile de pe raza și categoria dorită de acesta. Dacă nu sunt locații în acea zonă nu se va afișa nimic. Raza de căutare este calculată pe back-end în metri.

Celălalt buton din drop-down va activa interfața ce utilizează celălalt microserviciu al aplicației ce utilizează librăria open source GeoTools. Aici utilizatorul

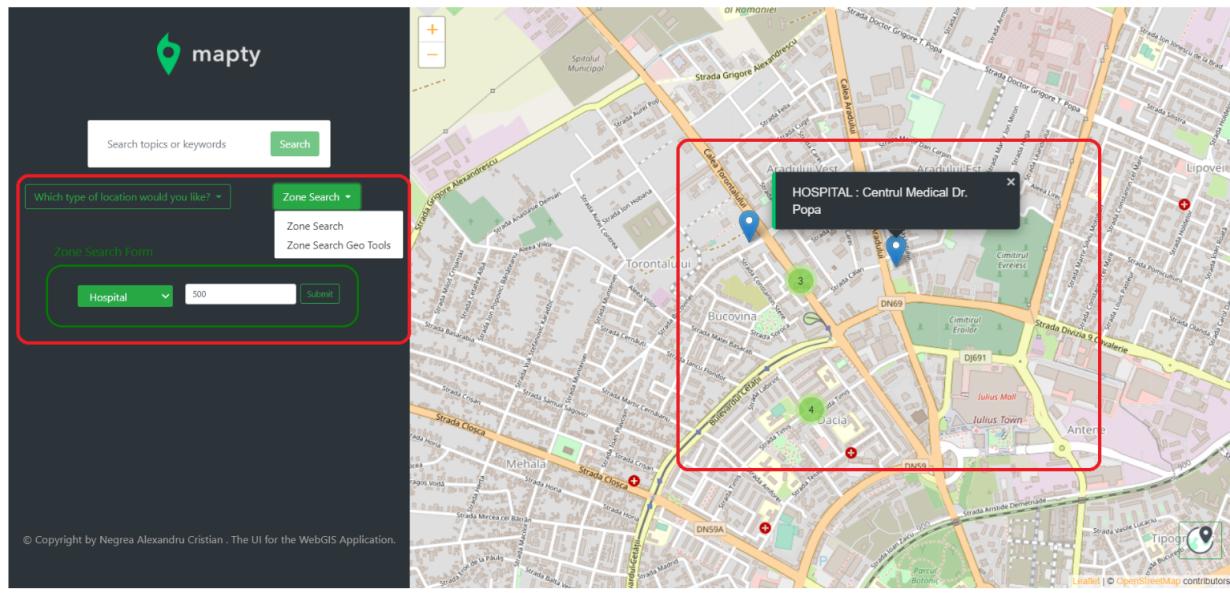


Figura 3.6: Zone search location folosind baza de date PostgreSQL.

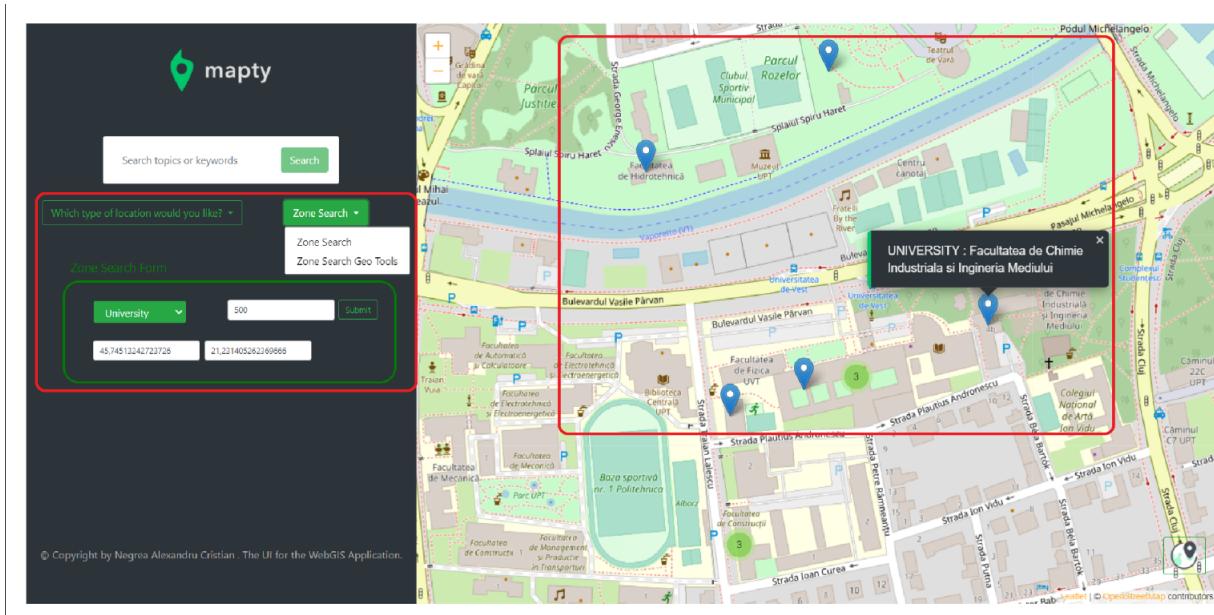


Figura 3.7: Zone search location folosind microserviciul ce utilizează librăria GeoTools.

va putea să returneze la fel, locațiile dintr-o zonă dată, dar aici utilizatorul va avea posibilitatea să aleagă o altă locație de pe hartă ca punct de reper sau să introducă manual coordonatele.

3.1.4 Get Preferred Locations

- **Get Preferred Locations:** în funcție de dorințele utilizatorului se vor returna toate locațiile din orașul respectiv dintr-o categorie dată de user, acestea fiind : Public, Medical și Transport. Deoarece acestea pot fi în număr mare se vor face cluster de aceste locații însenmând că se vor crea buline de culori diferite reprezentând numărul de locații din acea zonă, acestea pot fi observate și în câteva din use case-urile de mai sus. O dată apăsat un cluster se va da zoom pe hartă în zona dorită. Zonele unde sunt doar câteva locații se vor afișa markere normale iar, dacă încă există un număr mare de locații într-un punct se va crea un alt cluster pentru acelea.

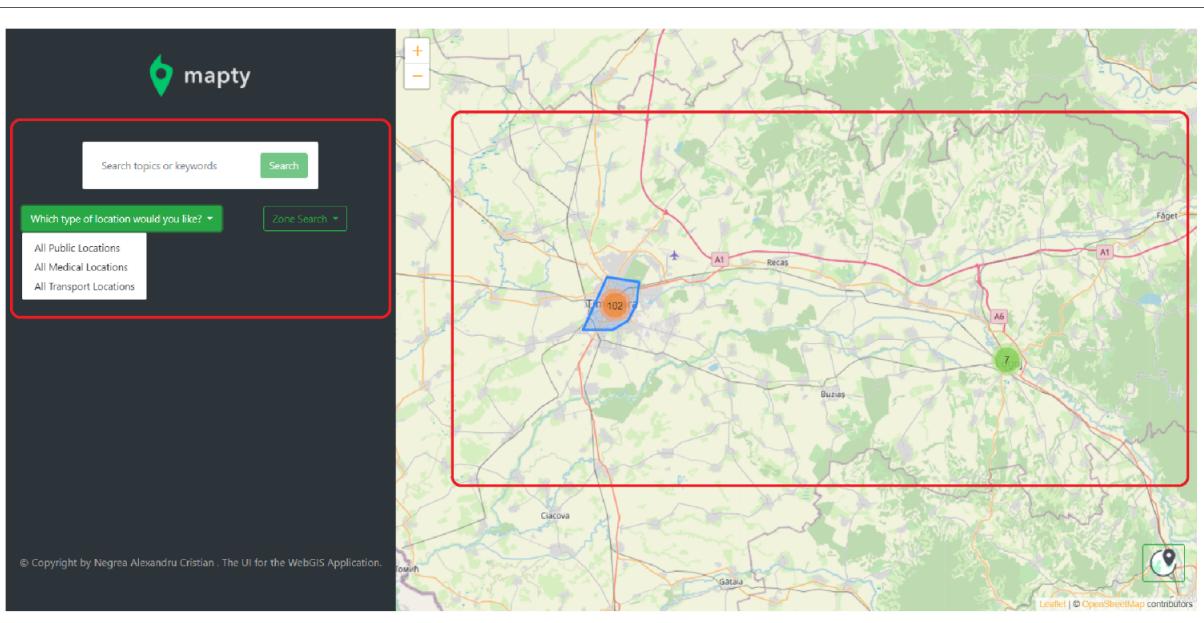


Figura 3.8: Toate locațiile din acea categorie.

În stânga, user-ul are posibilitatea să aleagă din mai multe categorii de locații. O dată selectată o categorie, după cum se vede și în imaginea de mai sus, aplicația va încerca să cuprindă toate locațiile ce există creănd clustere, astfel având performanță mai mare. Aceste acțiuni se realizează apelând endpoint-urile corespunzătoare fiecărei categorii de locații.

/

3.1.5 Calculate Distance

- **Calculate Distance:** aşa cum îi spune şi numele, calculează distanţa dintre două puncte (locaţii pe hartă) oferind un traseu optim (sau două, în funcţie de situaţie) și desenând traseul corespunzător. De asemenea se va afişa un meniu care îi va arăta utilizatorului diferenţele direcţiei pentru traseul ales.

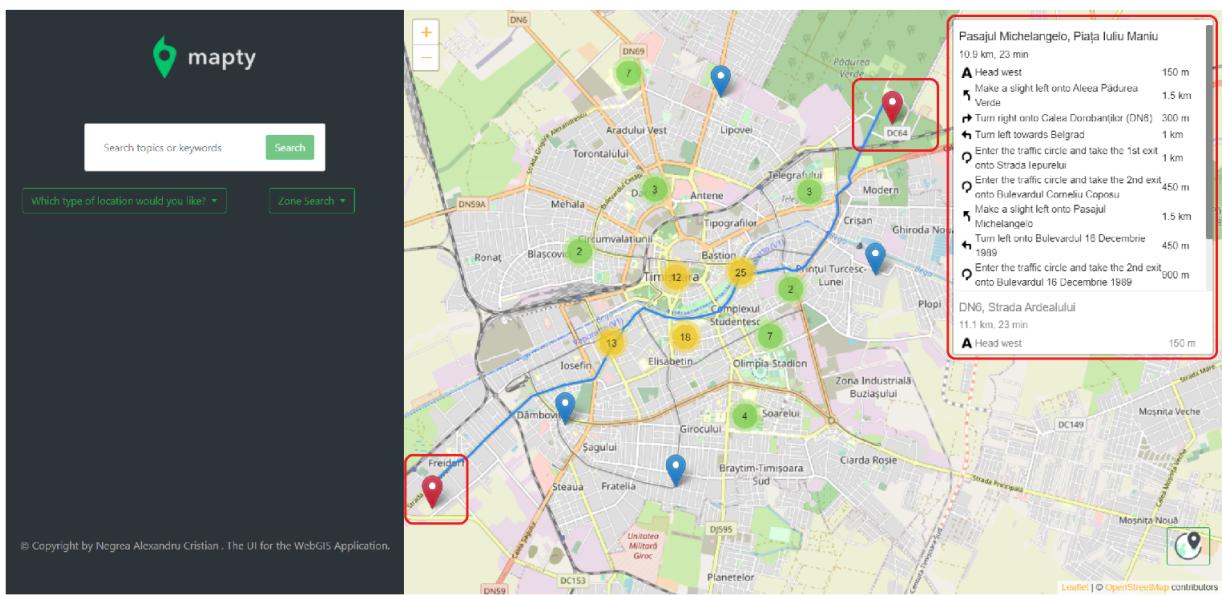


Figura 3.9: Distanţa dintre două locaţii specificate de utilizator.

În această imagine se observă un exemplu de calculare a distanței dintre două locații alese de user. Pentru a calcula distanța dintre două locații, utilizatorul va trebui să aleagă o locație dând click pe marker-ul cu locația dorită. Acest marker se va colora cu roșu însemnând că este selectat, alegând o a doua locație se va porni calcularea distanței afișând indicațiile ca în chenarul din dreapta sus după care se va observa cum traseul se desenează din punctul ales până la destinație. Acea interfață cu indicații poate fi folosită pentru a alege o rută secundară dând click pe textul colorat cu gri astfel recalculând ruta și redesenând traseul. De asemenea utilizatorul poate să vadă locul fiecărei indicații dând hover pe aceasta, iar pe hartă va apărea o bulină albastră indicând locația. Acțiunea de calculare a distanței poate fi intreruptă sau oprită selectând din nou unul dintre marker-ele selectate. Această calculare s-a realizat utilizând din librăria celor de la “Leaflet” serviciul lor de routing. Mai jos se află o parte din codul folosit pentru calcularea și afișarea distanței dintre două locații alese de utilizator.

```

1  private calculateDistanceBetweenTwoMarkers(): void {
2      ...
3      this.routingCalculator = L.Routing.control({
4          router: L.Routing.mapbox(Utils.MAP_BOX_API_KEY, {}),
5          // @ts-ignore
6          lineOptions: {
7              styles: [{className: 'animate'}] // Adding animate
8                  ↵ class
9          },
10         waypoints: [
11             L.latLng(startLat, startLng),
12             L.latLng(stopLat, stopLng)
13         ],
14     }

```

3.2 Scenarii reale de utilizare

În această secțiune se vor enumera mai multe scenarii reale de utilizare unde un utilizator poate folosi use case-urile descrise mai sus pentru rezolvarea unor probleme de orientare, căutarea unei locații, verificarea unei zone etc.

- Un utilizator poate dorește să se mute într-o anumită zonă din oraș unde să aibă mai multe zone publice de locul său de muncă cât și farmacii în zona unde locuiește. Acesta poate utiliza endpoint-ul numit “Get Zone Locations” sau “Get Locations” unde utilizatorul alege o locație dată de el sau din locația sa curentă și cauță toate locațiile pe raza dată și de tipul locației dorite. Folosind unul dintre cele două endpoint-uri va primi rezultatul dorit în formatul reprezentat mai sus în use case-ul numit “Get Zone Locations și Get Locations”.
- Un alt scenariu real de utilizare ar putea fi că un utilizator se află într-un oraș necunoscut și nu știe foarte bine zona și ar vrea să viziteze mai multe zone publice cum ar fi: școlile/universitățile. Pentru asta utilizatorul poate să folosească interfața descrisă mai sus pentru a afișa toate locațiile publice din orașul respectiv. De asemenea dacă acesta dorește să vadă distanța de la locația unde se află până la locația publică dorită, poate selecta cele două locații și se va calcula distanța cât și se va desena un traseu optim.
- Un alt bun scenariu de utilizare real ar putea fi: utilizatorul se află într-o zonă necunoscută de el și dorește să ajungă la cea mai apropiată locație cum ar fi: far-

macie, spital sau dorește să ajungă la o universitate din cealaltă parte a orașului și nu știe care este cea mai bună cale de a ajunge la locația dorită în cel mai optim timp posibil. Acesta ar putea folosi use case-ul numit “Calculate Distance“ care va calcula distanța dintre locația unde se află actual și locația dorită, iar aplicația va calcula această distanță. Aplicația va oferi atât suport vizual unde utilizatorul va putea vedea calea trasată, va avea o listă ce conține toate indicațiile și va putea da hover pe ele pentru a vedea exact locul indicației cât și posibilitatea unei rute alternatorie dacă ruta selectată nu este cea dorită.

În final, Web Map Service-ul este doar demonstrativ și a fost creat pentru a exemplifica o utilizare a Web Feature Service-ului creat și de a arăta folosința acestuia pe un caz real de utilizare.

Capitolul 4

Concluzii și probleme deschise

Aplicația creată în această lucrare este un “Web Feature Service“ ce respectă standardele “OGC“ folosind răspunsuri în formatul “GeoJSON“ fiind structurată în două microservicii ce au implementări diferite dar returnează răspunsuri identice. Această aplicație de tipul “RestAPI“ oferă vectori de geometrii ce pot fi folosite de aplicații pentru a crea ideea de “Smart City“ ce reprezintă una dintre soluțiile asupra provocărilor în creșterea urbanizării.

Aplicația a fost realizată folosind framework-ul popular numit “Spring“ utilizând “Spring Boot“ având la bază limbajul orientat obiect “Java“. Aplicația, cum a fost construită folosind două microservicii ce folosesc abordări diferite de manipulare a datelor, oferă clientului posibilitatea să folosească oricare dintre cele două ce îi satisfac nevoile de utilizare.

Pentru exemplificarea unui scenariu de utilizare, s-a creat o aplicație demonstrativă construită în framework-ul celor de la “Google“, “Angular“ utilizând NgRx ce aduce un “reactive state management“ aplicației. În cadrul acesteia s-au utilizat use case-urile aplicației back-end pentru a exemplifica aceste utilizări folosind ambele microservicii puse la dispoziție. Ambele aplicații au fost exemplificate și explicate în capitolele de mai sus, Capitolul 2, respectiv 3.

Crearea aplicației propriu-zise, propusă pentru lucrare, a necesitat multe ore de studiu și analiză a unei arhitecturi optime, învățarea și netezirea framework-urilor utilizate cât și testarea și încărcarea acesteia pe “AWS Cloud“. Cu toate acestea am dobândit experiență în utilizarea framework-ului Spring, astfel ajungând la un nivel mai ridicat de înțelegere și utilizare al acestuia, o aprofundare a limbajului “Java“ cât și a cunoștințelor “SQL“ și crearea și manipularea datelor spațiale utilizând “PostGreSQL“ cât și extensia sa “PostGIS“, dar și învățarea și punerea bazei pentru framework-ul popular celor de la “Google“, “Angular“ ce se îmbină destul de bine cu preferințele mele.

Posibile probleme întâmpinate pe parcursul creării aplicației au fost minimale, nimic ce nu s-a putut rezolva cu ușurință, cele mai comune fiind problemele de conexiune dintre back-end (WFS) și baza de date, configurarea microserviciilor să comunice cu serverul de proxy cât și comunicarea cu front-endul (WMS).

Totuși, datorită faptului că bază de date nu este legată de o instanță cu mare putere de procesare pe “AWS“ ar putea exista probleme de latență în comunicarea cu “Web

Feature Service-ul“. Din partea aplicației “RestAPI“ problemele de latență ar trebui să fie aproape inexistente, deoarece se folosește o instanță “EC2“ cu resurse mai bune. S-a încercat împărțirea fiecărei instanțe de “Spring Boot“ pe multiple instanțe cu putere de procesare mult mai mică, dar s-a dovedit a fi imposibil deoarece conexiunea dintre “Pr‘oxy Server“ și cele două microservicii nu se putea realiza, aşadar s-a decurs la crearea unei singure instanțe cu suficiente resurse ce este capabilă să poată țină în picioare toate instanțele necesare rulării optime a aplicației.

Aplicația în sine, oferă endpoint-uri ce sunt utile pentru un client ce dorește să conecteze acest “Web Feature Service“ de “Web Map Service-ul“ sau și să obțină funcționalități ce ar putea fi folosite în crearea unei aplicații utile.

De asemenea există loc de îmbunătățiri cum ar fi:

- Adăugarea de mai multe date pentru a oferi funcționalitate în multiple orașe din România și de pretutindeni.
- Integrarea de noi funcționalități de calculare a distanțelor dintre două locații nu doar din punct de vedere matematic, cât și geografic ținând cont de străzi, trotuare etc
- Adăugarea de mai multe microservicii ce pot fi utile client, împărțind funcționalitățile pe mai multe microservicii, astfel făcând aceste microservicii modulare și ușor de decuplat în cazul în care nu mai este nevoie de acestea
- Adăugarea unui serviciu de logare astfel salvând căutările, locațiile cele mai frecventate și crearea de rute custom ce ține cont de preferințele fiecărui utilizator
- etc

Bibliografie

- [1] John Carnell *Spring Microservices in Action Second Edition.* Manning Shelter Island 2017
- [2] Flyway
- [3] Spring-Cloud
- [4] <https://www.ogc.org/standards/wfs> - WFS
- [5] <https://datatracker.ietf.org/doc/html/rfc7946> - GeoJSON
- [6] <https://www.ogc.org/standards> - OGC
- [7] <https://aws.amazon.com/s3/> - Amazon S3 Bucket
- [8] <https://aws.amazon.com/rds/> - Amazon RDS Database
- [9] <https://aws.amazon.com/ec2/?ec2-whats-new.sort-by=item.additionalFields.postDateTime&ec2-whats-new.sort-order=desc> - Amazon EC2
- [10] <https://cloud.spring.io/spring-cloud-netflix/reference/html> - Eureka
- [11] <https://spring.io/> - Spring
- [12] <https://spring.io/projects/spring-cloud-netflix> - Spring Cloud Netflix
- [13] <https://spring.io/projects/spring-boot> - Spring Boot
- [14] <https://flywaydb.org/documentation> - Flyway
- [15] <https://docs.spring.io/spring-boot/docs/2.1.18.RELEASE/reference/html/boot-features-validation.html> - Validation
- [16] <https://spring.io/projects/spring-data-jpa> - Spring Data JPA
- [17] <https://projectlombok.org/> - Lombok
- [18] <https://swagger.io/> - Swagger
- [19] <https://geotools.org/> - GeoTools
- [20] <https://hibernate.org/> - Hibernate
- [21] <https://angular.io/docs> - Angular Framework

- [22] <https://www.openstreetmap.org/#map=4/43.49/26.87> - *Open Street Map*
- [23] <https://developers.arcgis.com/rest> - *ArcGIS Rest API*
- [24] [https://www.bentley.com/en/products/product-line/
asset-performance/opencities-map-enterprise](https://www.bentley.com/en/products/product-line/asset-performance/opencities-map-enterprise) - *Bentley Map Enterprise*
- [25] <https://www.hexagongeospatial.com/products/power-portfolio/geomedia>
- *Bentley Map Enterprise*
- [26] <http://geoserver.org> - *GeoServer*